Almacenamiento de Datos.

En esta unidad de trabajo vamos a descubrir la importancia de ser capaces de almacenar datos de forma distribuida como ingrediente fundamental para nuestros sistemas Big Data.

Comenzamos viendo qué es un sistema de ficheros, para después descubrir la existencia de sistemas de almacenamiento que funcionan de modo distribuido entre nodos de un clúster y de sistemas que de forma específica realizan esa tarea no con unidades de almacenamiento sino en memoria.

Más adelante conoceremos *HDFS*, el sistema de almacenamiento distribuido de *Hadoop*.

A continuación haremos un repaso por el mundo de las bases de datos *NoSQL*,descubriendo tanto algunos conceptos básicos en relación a las mismas como qué tipos existen.

Por último seguiremos el tutorial oficial de *MongoDB*, la cual es una de las bases de datos *NoSQL* más conocidas y utilizadas.

1. Sistemas de Ficheros.

Un sistema de ficheros es la infraestrucutra necesaria para poder almacenar y organizar datos en unidades de almacenamiento, ya sean discos duros, memorias de estado sólido,cintas o *DVD*s.

Para ello, se utiliza el concepto de fichero como la unidad atómica de almacenamiento empleada para almacenar los datos.

Gracias al sistema de ficheros podemos tanto guardar como leer ficheros de forma independiente, considerando se uno de los primeros pasos a la hora de organizar los datos de modo que comiencen a considerarse como información.

Además, el sistema de ficheros mantiene una estructura en forma de árbol (el árbol de directorio) que nos permite dar un paso más en la organización de la información, almacenando cada fichero en una carpeta según su significado o el uso que queramos darle.

El sistema operativo es capaz de interactuar con el sistema de ficheros, gracias a lo cual puede ofrecer servicio de almacenamiento y recuperación de datos a las aplicaciones que en él se ejecutan.

Hay que tener en cuenta que aunque hablemos en singular, realmente existen diversos tipos de sistema de fichero, como ejemplo FAT32, NTFS o EXT4. Los sistemas operativos suelen emplear sólo uno de ellos para almacenar su propia información pero a la vez suelen ofrecer compatibilidad con otros para poder acceder a datos auxiliares o externos.

Es importante que sepas qué es un sistema de ficheros (también llamado sistema de archivos).

Puedes encontrar más información en el siguiente enlace:

Sistema de archivos

Puedes ver más información sobre lo que es un sistema operativo en el siguiente enlaces:

Sistema operativo

En los siguientes enlaces puedes ver más información sobre los sistemas deficheros que hemos mencionado:

Tabla de asignación de archivos (FAT12, FAT16, FAT32).
NTFS
EXT4

1.1.- RAID.

RAID es un sistema de almacenamiento de datos que es capaz de distribuirlos por múltiples unidades según diversas estrategias para así conseguir o bien unidades mayores que el mayor de los discos que tengamos, o bien replicación automática de nuestros datos o bien ambas cosas.

Tengamos aquí en cuenta que RAID realmente no es de por sí un sistema de ficheros sino una capa que trabaja por debajo de ellos. Por ejemplo podemos tener dos discos duros de 2TB conectados mediante RAID de modo que el sistema de ficheros no ve esas dos unidades sino una única unidad de 4TB.

RAID puede trabajar:

- Por hardware, mediante controladoras RAID específicas.
- Por software, haciendo que el procesador ejecute un software que realiza el trabajo equivalente al que haría la controladora hardware (lo cual es más lento pero más barato).

Como ya hemos dicho, RAID permite configurar los discos según distintas estrategias, llamadas niveles.

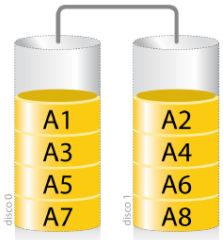
Los niveles más empleados son los llamados RAID 0, RAID 1 y RAID 5.

RAID 0: Conjunto dividido.

Podemos ver dos discos de 1TB como uno de 2TB sin replicación.

- Permite ver unidades virtuales más grandes que las unidades físicas.
- Sin redundancia.
- Aproximadamente el doble de velocidad en lectura y escritura.

Diagrama de unaconfiguración RAID 0

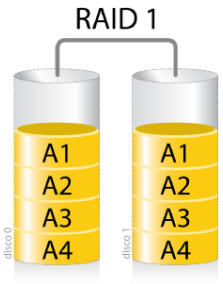


RAID 1: Conjunto en espejo.

Podemos ver dos discos de 1TB como uno de 1TB con replicación.

- Permite tener los datos replicados de forma automática.
- Mayor seguridad desaprovechando capacidad.
- Hasta el doble de velocidad de lectura. Sin ganancia para escritura.
- Tolerante al fallo de uno de los discos.

Diagrama de unaconfiguración RAID 1

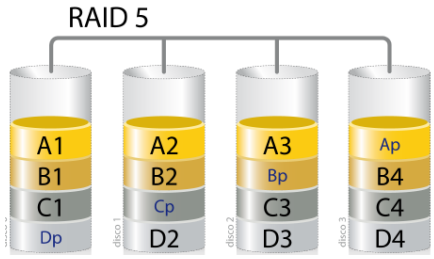


RAID 5: Conjunto dividido con paridad distribuida.

Podemos ver tres discos de 1TB como uno de 2TB con replicación.

- Es necesario un mínimo de 3 discos.
- Permite ver unidades virtuales más grandes que las unidades físicas añadiendo además replicación.
- Permite mayores rendimientos de lectura al poder leer de varias unidades a la vez.
- Tolerante al fallo de uno de los discos.

Diagrama de una configuración RAID 5



Puedes ver más información sobre RAID en el siguiente enlace:

RAID

1.2.- Sistemas de ficheros distribuidos.

Un sistema de ficheros distribuido es aquel que es capaz de almacenar ficheros distribuyendo el contenido de los mismos en las distintas máquinas que conforman un clúster. Permite interactuar con ellos de un modo unificado, como si realmente residiesen en la máquina en la que se ejecuta la aplicación que va a utilizarlos.

Los sistemas de ficheros distribuidos que se emplean para sistemas Big Data permiten almacenar ficheros más grandes que el espacio de almacenamiento disponible en cualquiera de los nodos del clúster. Ello se consigue gracias a que los ficheros no se almacenan completos sino divididos en fragmentos (llamados bloques), de modo que se puede almacenar ficheros enormes simplemente distribuyendo los bloques que lo conforman por el clúster.

Estos sistemas son agnósticos respecto de los datos que van a almacenar, y proporcionan redundancia y alta disponibilidad gracias a la replicación en distintos nodos de los bloques que componen los ficheros. Tienen un indice de replicación configurable, de modo que cada bloque queda al menos en N nodos distintos (donde N debe ser como mínimo 2 para que haya replicación).

Esta capacidad de replicación integrada dentro del propio sistema hace que ya no sea necesario contar con replicación (tipo RAID) dentro del sistema de ficheros local de cada nodo.

Debido a que cada fichero se puede encontrar distribuido en distintos nodos, se pueden alcanzar tasas de lectura y escritura superiores a las de las unidades de almacenamiento individuales. Ello se debe a que tales lecturas y escrituras se realizan en paralelo, evitando el cuello de botella que se produce al tener que emplear una única unidad física.

Por razones de diseño, los sistemas de ficheros distribuidos para Big Data emplean bloques muy grandes (por ejemplo de 128MB). Por esa razón, no son ideales para trabajar con muchos ficheros pequeños sino que funcionan mejor con pocos ficheros de tamaño grande. Debido a esto, cuando los datos de las fuentes vienen en muchos ficheros pequeños lo que se hace es combinarlos en ficheros grandes durante el proceso de ETL.

Uno de los sistemas de ficheros distribuidos más conocidos y empleados en entornos BigData es HDFS (de Apache Hadoop), el cual tiene como precursor a GFS (de Google).Dentro del mundo cloud, el más conocido es Amazon S3, aunque existen otras alternativas como Google Cloud Storage y Azure Blob Storage.

Es importante que sepas qué es un sistema de archivos distribuido.

Puedes encontrar más información en el siguiente enlace:

Sistema de archivos distribuido

En el los siguientes enlaces puedes encontrar más información sobre algunos de los sistemas de ficheros distribuidos que hemos mencionado. No incluimos HDFS porque en esta unidad de trabajo contamos con una sección específica para el sistema de ficheros distribuido de Hadoop.

Google File Sistem (GFS)

Amazon S3

1.2.1.- Almacenamiento distribuido en memoria.

Con el precio de la memoria RAM cada vez más baja y la posibilidad de incluir cada vez más cantidad en cada máquina individual cada vez es más común el uso de sistemas de almacenamiento distribuido en memoria para Big Data.

Mantener los datos en memoria RAM elimina el gran problema de latencia en entrada-salida de las unidades de almacenamiento y del tiempo de transferencia entre el almacenamiento y la memoria.

Para hacernos una idea, la lectura secuencial de 1 MB de datos desde un almacenamiento de disco puede requerir alrededor de 20ms, mientras que para la misma lectura desde memoria RAM se necesita del orden de 0.2ms (evidentemente todo ello dependiendo de qué almacenamiento y qué memoria).

Una de las ventajas más claras de usar un almacenamiento en memoria es que permite enfrentarse con mayor rendimiento al rápido flujo de datos de entrada al sistema Big Data (la característica de velocidad). Además, gracias a que podemos obtener velocidades de procesamiento mucho mayores, se pueden realizar analíticas en tiempo real más complejas y/o producir descubrimientos en los datos mucho más rápido para así ser capaces de tomar buenas decisiones de negocio en un menor tiempo.

Contando con que los nodos tienen instalada ya toda la memoria RAM que su placa base permite, para conseguir más memoria en el sistema es necesario realizar un escalado horizontal añadiendo más nodos.

El hecho de necesitar añadir un nodo completo cada vez que queremos aumentar tanta memoria como éste pueda tener instalada hace que tales ampliaciones resulten costosas.

El mayor problema de los almacenamientos en memoria es que no proporcionan durabilidad de por sí (los datos se pierden si se apaga la máquina). Por esa razón, estos almacenamientos en muchas ocasiones se utilizan sólo para datos que vamos a utilizar para algún tipo de analítica y vienen copiados desde otro almacenamiento que sí es persistente.

Un almacenamiento en memoria es apropiado cuando:

- Los datos llegan rápidamente.
- Se requiere analítica continua y/o en tiempo real o en streaming.
- Es necesario realizar consultas interactivas.
- Es necesario poder visualizar datos en tiempo real.
- El mismo conjunto de datos se utiliza a la vez para varias tareas.
- Es necesario poder acceder de forma iterativa al mismo conjunto de datos sin necesidad de volver a cargarlos desde disco cada vez.
    - Análisis de datos exploratorio o iterativo.
    - Algoritmos basados en grafos.
- Hay que desarrollar soluciones Big Data de baja latencia con que soporten transacciones *ACID*.

Un almacenamiento en memoria no es adecuado cuando:

- El procesamiento es por lotes.
- Necesitamos trabajar con grandes cantidades de datos.
- La persistencia de datos según llegan desde sus fuentes es una necesidad.
- El presupuesto es limitado y preveemos que vamos a necesitar ampliar memoria con nuevos nodos.

1.3.- Hadoop Distributed File System(HDFS).

HDFS (Hadoop Distributed File System) es uno de los sistemas de ficheros distribuidos para Big Data más conocidos y usados, gracias a formar parte de Hadoop, la cual es la plataforma Big Data de código abierto pionera.

En la práctica, HDFS es una versión de código abierto del sistema de ficheros distribuidos de Google (GFS), ya que se basa en un artículo científico publicado por Google en el año 2003 en el que se detalla el funcionamiento de GFS.

Está diseñado para ser capaz de almacenar y gestionar una gran cantidad de datos incluyendo ficheros de gran tamaño, distribuyendo los por un clúster cuyos nodos son *commodity hardware*, (lo cual hay que tener en cuenta que no equivale a máquinas baratas).

Una de sus características importantes es que como cualquier sistema de ficheros (y a diferencia de las bases de datos relacionales), HDFS no obliga a que los datos sean almacenados cumpliendo un determinado esquema establecido de antemano. Esta característica, conocida como *schema-on-read* (por oposición a la característica *schema-on-write* de las bases de datos relaciones), ahorra trabajo manteniendo o modificando el esquema con el que se almacenan los datos.

Dado que permite guardar datos en nodos de precio reducido, se suele pensar en HDFS como un "disco duro enorme que sale barato".

Gracias ello y a que es *schema-on-read*, una estrategia muy común es guardar en HDFS, según llegan, todos los datos que consideramos que pueden tener algún valor, a la espera de ver más adelante cómo serán interpretados para obtener dicho valor.

Puedes ver más información sobre HDFS en el siguiente enlace:

Hadoop Distributed File System

1.3.1.- Características y funcionamiento.

Para alcanzar sus objetivos, HDFS particiona los ficheros en bloques de gran tamaño (128MB por defecto, aunque ese valor es configurable). La razón de tal gran tamaño es que persigue minimizar el número de búsquedas en disco para así poder trabajar a la velocidad de transferencia de los mismos. Hay que tener en cuenta que cada nodo almacena los bloques haciendo uso de su propio sistema de ficheros local. Lo que se quiere por lo tanto es que los nodos no tengan que buscar en sus discos muchos ficheros pequeños sino pocos ficheros grandes para así poder hacer lecturas de contínuo.

Esta característica implica que HDFS no está optimizado para latencia sino para velocidad de transferencia. Ello en parte también se debe a que Hadoop fue concebido para realizar analítica por lotes en lugar de en tiempo real.

Es importante tener en cuenta que a pesar de que los bloques tengan un tamaño de 128 MB, eso no implica que un fichero HDFS más pequeño de tal cantidad desperdicie espacio de almacenamiento (el fichero local en el que el nodo guarda el bloque puede ocupar menos de128 MB).

Al margen de que los bloques sean grandes, HDFS también prefiere que los ficheros que almacena (aquí nos referimos a los propios ficheros HDFS, no a los bloques individuales) también sean grandes. Ello se debe a que para conseguir una mayor velocidad de acceso el directorio se mantiene en la memoria de uno de los nodos (el *Namenode*), por lo que una gran cantidad de ficheros pequeños necesitaría una memoria RAM muy grande.

Gracias a la distribución de cada fichero en bloques por el clúster, cada fichero puede ser más grande que la mayor unidad física de almacenamiento.

Otra característica interesante de HDFS es la llamada "rack awareness", según la cual se guarda constancia de cuál es la topología de conexiones (via switch) de los nodos del clúster.

De este modo al ejecutar un trabajo se puede realizar una distribución del mismo de modo que se maximice la localidad de los datos y se minimice el número de saltos entre switch que los mismos tienen que realizar.

Tengamos aquí en cuenta que a pesar de que generalmente hay una equivalencia entre cómo están dispuestos los nodos en racks y sus conexiones a switch, en este caso lo realmente importante no es el rack en el que está el nodo sino el switch al que se conecta.

1.3.2.- Acceso mediante línea decomandos.

HDFS es un componente de Apache Hadoop, por lo que el modo estándar para poder ponerlo a funcionar (ya sea en una única máquina o en varias) es realizar una descarga de Hadoop y tras eso hacer la correspondiente instalación.

Descarga de Hadoop

Sin embargo, también es cierto que dado que Hadoop realmente está compuesto por una serie de procesos java ejecutándose en los distintos nodos del clúster, técnicamente podríamos poner HDFS a funcionar de forma independiente arrancando sólo los correspondientes procesos namenode y datanode en los nodos que consideremos oportuno.

Podemos acceder a HDFS para escribir y leer ficheros a través de línea de comandos en cualquier máquina que tenga instalado Hadoop si ejecuta el servicio de HDFS (sin importar si se trata de un namenode o de un datanode). El acceso es muy similar al que se realiza en un sistema Unix/Linux.

Mediante el comando *hadoop* y usando *fs* como primer parámetro, podemos realizar las funciones más básicas como mostramos a continuación. Debe tenerse en cuenta que HDFS emplea un árbol de directorio con ficheros y permisos al estilo Unix/Linux, y que por defecto se emplea el directorio raíz '/'.

Listado de ficheros:

Podemos listar ficheros de un modo muy similar a cuando utilizamos ls -l en Unix/Linux.

```
$ hadopp fs -ls <ruta>
```

Creación de directorios:

Similar al comando mkdir de Unix/Linux.

```
$ hadoop fs -mkdir <ruta>
```

Copia de ficheros del sistema de ficheros local a HDFS:

En este caso la copia es similar a cuando usamos el comando copy en Unix/Linux, pero con la diferencia de que para copiar desde local a HDFS usaremos copyFromLocal.

```
$ hadoop fs –copyFromLocal <origen_local> <destino>
```

Copia de ficheros desde HDFS al sistema de ficheros local:

En este caso copyToLocal.

```
$ hadoop fs –copyToLocal <origen> <destino_local>
```

Impresión del contenido de un fichero:

Muy similar a como funcionan los comandos cat y tail en Unix/Linux.

```
$ hadoop fs –cat <ruta_fichero>
```

... para imprimir el fichero completo.

```
$ hadoop fs –tail <ruta_fichero>
```

... para imprimir el último kB del fichero.

Copiar un fichero de una ruta a otra dentro de HDFS:

Similar al comando cp de Unix/Linux.

$ hadoop fs –cp <origen> <destino>

Mover un fichero de una ruta a otra dentro de HDFS:

Similar al comando mv de Unix/Linux.

$ hadoop fs –mv <origen> <destino>

Eliminar un fichero de HDFS:

Similar al comando *rm* de Unix/Linux:

$ hadoop fs –rm [-r] <ruta>

En este caso el parámetro *-r* borraría dentro de directorios de forma recursiva.

Acceder a la ayuda para ver más comandos:

$ hadoop fs –help

Ten en cuenta que además de usar *hadoop fs* podemos usar *hdfs dfs* y en tal caso el comportamiento es ligeramente diferente.

Puedes ver más información sobre cómo interactuar mediante *hdfs dfs* en el primer capítulo del siguiente libro disponible online desde la web de O'Reilly.

Hadoop with Python by Zach Radtka, Donald Miner (capítulo 1)

1.3.3.- Acceso mediante Python.

Existen diversos paquetes o librerías en diversos lenguajes de programación que permiten acceder a HDFS de forma programática, gracias a lo cual podemos conseguir una interacción más rica que por línea de comandos.

En el caso de Python existen distintas opciones.

Nosotros vamos a centrarnos en cómo se haría usando **Snakebyte**.

Snakebyte es un paquete de Python que permite acceder a HDFS. Implementa el protocolo Hadoop RPC, para acceder al Namenode sin necesitar hacer llamadas a hadoop fs o hdfsdfs.

Vamos a basarnos en el primer capítulo del siguiente libro disponible online desde la web de O'Reilly.Puedes verlo en el siguiente enlace:

Hadoop with Python by Zach Radtka, Donald Miner (capítulo 1)

Veamos cómo podemos acceder a HDFS desde Python empleando Snakebite. (vamos a dar por hecho que ya están correctamente instalados).

Listado del contenido de un directorio:

Contenido del fichero listar_directorio.py

```
from snakebite.client import Client
```

```
client = Client('localhost', 9000)
for x in client.ls(['/']):
print x
```

```
$ python listar_directorio.py
```

```
{"Grupo": u'supergroup', 'permission': 448, 'arch': 'choque': 'd', 'access-
time': 0L, 'block-replication: 0, 'modification-time': 1442752574936L,
'longitud': 0L, 'blocksize': 0L, 'owner': u'hduser', 'path': '/tmp'}.

{"'group': u'supergroup'", 'permiso': 493, 'file': 'd', 'access-time': 0L,
'block-replication': 0, 'modificación-tiempo': 1442742056276L, 'longitud': 0L,
'bloquesize': 0L, 'owner': u'hduser', 'path': '/usuario'}.
```

Crear un directorio:

Contenido del fichero *crear_directorio.py*

```
from snakebite.client import Client
client = Client('localhost', 9000)
for p in client.mkdir(['/foo/bar', '/input'], create_parent=True):
print p
```

```
$ python crear_directorio.py
```

```
{'path': '/foo/bar', 'result': True}
{'path': '/input', 'result': True}
```

Eliminar ficheros y directorios:

Contenido del fichero eliminar.py

```
from snakebite.client import Client
client = Client('localhost', 9000)
for p in client.delete(['/foo', '/input'], recurse=True):
print p
```

```
$ python eliminar.py
```

```
{'path': '/foo', 'result': True}
{'path': '/input', 'result': True}
```

Copiar ficheros de HDFS al sistema de ficheros local:

Contenido del fichero *copiar_a_local.fs*

```
from snakebite.client import Client
client = Client('localhost', 9000)
for f in client.copyToLocal(['/input/input.txt'], '/tmp'):
print f
```

```
$ python copiar_a_local.fs
```

```
{'path': '/tmp/input.txt', 'source_path': '/input/input.txt', 'result': True,
'error': ''}
```

Lectura de ficheros de HDFS:

Contenido del fichero *texto.py*

```
from snakebite.client import Client
client = Client('localhost', 9000)
for l in client.text(['/input/input.txt']):
print l
```

```
$ python text.py

blanco
negro
azul
```

2.- Bases de datos NoSQL.

Las bases de datos relacionales están en todos los sitios y constituyen una infraestructura vital para gran infinidad de actividades transaccionales.

Sin embargo, algunas de las características que les otorgan grandes ventajas producen a su vez otros problemas.

Uno de ellos lo hemos visto en el caso práctico. Las bases de datos relacionales son *schema-on-write*, lo cual significa que necesitamos conocer el formato de los registros (qué atributos y de qué tipos) antes de escribirlos en las tablas. Eso obliga a un diseño previo del esquema en un momento en el que quizás no tenemos todo el conocimiento sobre el problema.

Evidentemente podemos añadir, eliminar y modificar el esquema de las tablas sobre la marcha, pero eso reviste ciertos peligros si la base de datos está en producción, siendo por lo tanto un quebradero de cabeza para quienes gestionan esas bases de datos.

Para solventar este problema existen bases de datos *schema-on-read*, que permiten realizar escrituras (en este caso no de registros sino de documentos) sin necesidad de cumplir con un esquema de tabla. Es en el momento de la lectura cuando se interpreta el esquema de cada documento.

Las bases de datos NoSQL ("no sólo SQL" o "no relacionales"), son un conjunto de bases de datos que por alguna razón escapan de la denominación de base de datos relacional.

Ello implica que por lo general no usan SQL como lenguaje de consulta, aunque esto no es una regla absoluta y de hecho la razón por la que a veces son llamadas "no sólo SQL" es porque algunas sí que pueden emplear lenguajes de tipo SQL.

En el siguiente enlace puedes ver más información sobre NoSQL.

NoSQL

2.1.- Conceptos generales.

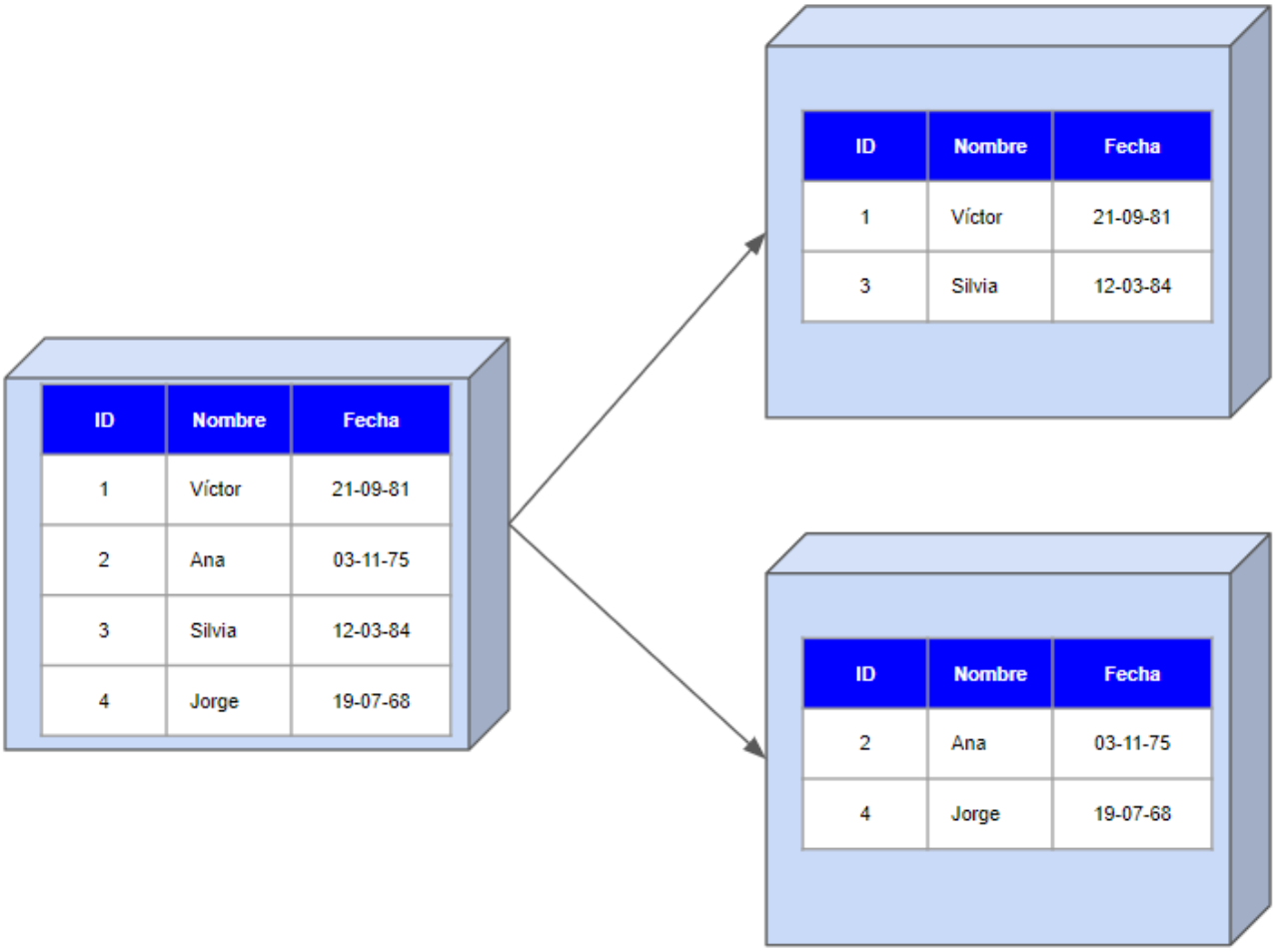
En esta sección veremos algunos conceptos generales acerca de las bases de datos distribuidas que se pueden aplicar al caso de *NoSQL*.

Veremos aquí un esquema/resumen para que puedas tener una vista general:

- ***Sharding*:**
  Dividir los conjuntos de datos para distribuirlos por un clúster.
- **Replicación:**
  Hacer copias de los datos en distintos nodos del clúster.
- ***Sharding* con replicación:**
  Combinar las dos características anteriores para obtener las ventajas de ambas.

Es importante destacar que tanto el *sharding* como la replicación no son en absoluto exclusivos de las bases de datos NoSQL, sino que se emplean en muchos otros sistemas. Tampoco todas las bases de datos NoSQL cuentan con estas características, aunque la mayor parte sí.

2.1.1.- Sharding.



El *sharding* es un mecanismo para particionar un conjunto de datos en subconjuntos más pequeños (*shards*) de modo que puedan ser distribuidos por los distintos nodos de un cluster.

Una de las posibilidades que ofrece este mecanismo es el ser capaz de tratar con conjuntos de datos más grandes de lo que sería capaz de almacenar cualquier máquina del clúster deforma individual.

También facilita la distribución de la carga de trabajo en los diversos nodos (trabajando cada uno sólo con una parte del conjunto de datos) para conseguir escalabilidad horizontal.

Otro beneficio del ***sharding*** es que proporciona tolerancia parcial a fallos en los nodos, ya que si uno queda inaccesible sólo resulta afectada la fracción de datos que en él reside.

Cuando se realiza una consulta es necesario primero acceder a todos los nodos que puedan contener datos a devolver y después fusionar todas las respuestas desde esos distintos nodos para producir la respuesta final. Sólo en el caso de que sólo uno de los nodos se vea afectado por la consulta (o si casualmente sólo uno de ellos devuelve algún resultado) se puede prescindir de este paso final.

El mayor problema de este funcionamiento es que debe diseñarse la estrategia de ***sharding*** (el algoritmo o la fórmula que determina qué registro va en qué *shard*) teniendo en cuenta cuáles van a ser las necesidades de acceso/consulta a los datos. Para ello, se ha de intentar que aquellos datos que suelan accederse juntos queden localizados en el mismo *shard*, lo cual muchas veces es complicado y en ocasiones (según el uso de la base de datos) imposible.

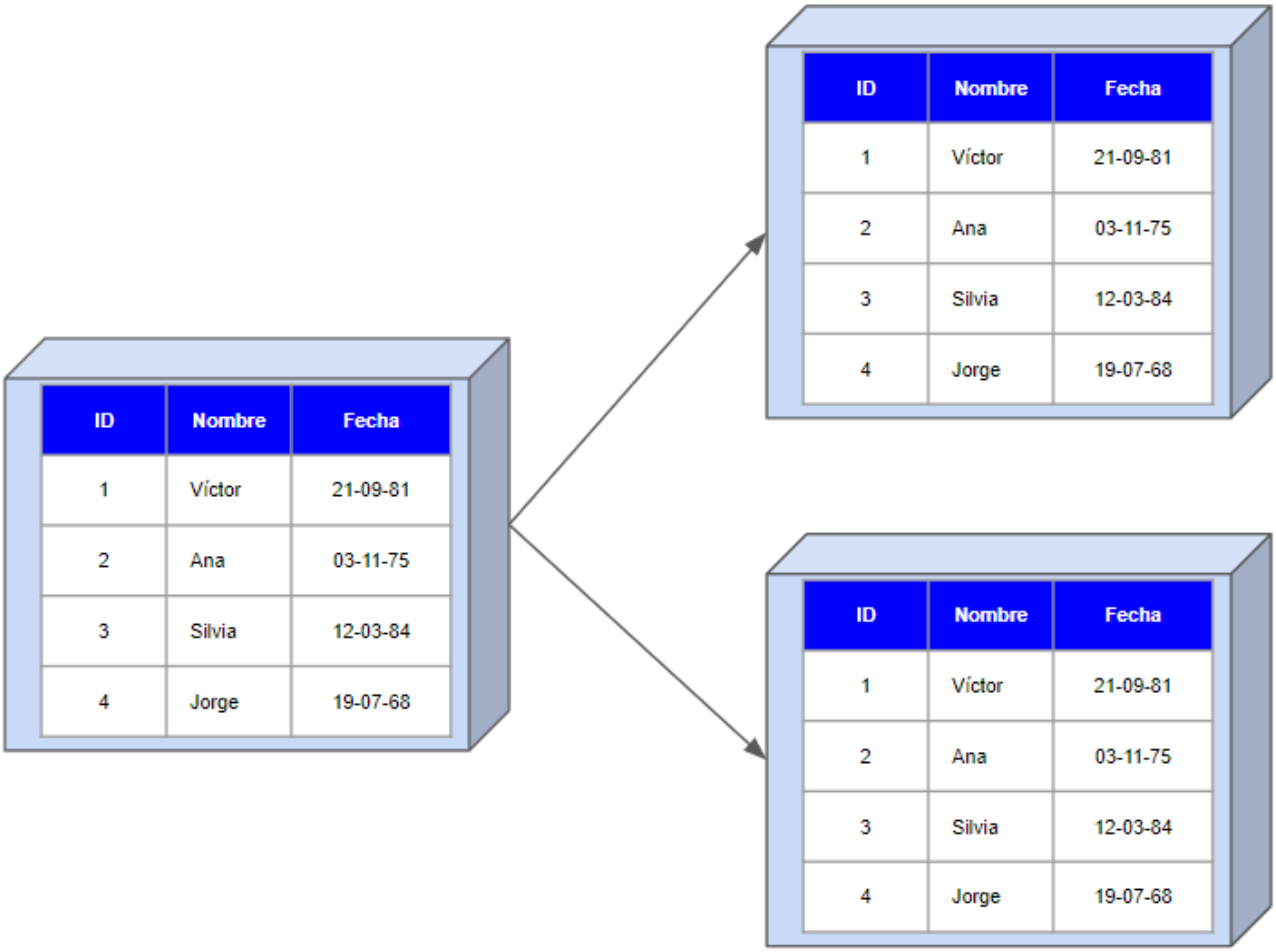
Puedes ver más información sobre lo que es *sharding* en el siguiente enlace:

Shard (arquitectura de base de datos)

Dado que el contenido de ese enlace en castellano es exiguo, recomendamos acceder a su versión en inglés:

Shard_(database_architecture)

2.1.2.- Replicación.



La replicación cosiste en almacenar copias de un mismo conjunto de datos (llamadas réplicas) en distintos nodos de un clúster.

Gracias a ello se consigue mayor disponibilidad (se puede leer desde distintos nodos) y tolerancia a fallos (si un nodo no está accesible, la información que guarda sigue disponible en otro/s).

Existen dos estrategias distintas a seguir para implementar el mecanismo de replicación:

Maestro-esclavo (*master-slave*):

- Todas las escrituras (inserciones, actulizaciones y borrados) se realizan en el nodo maestro y después son replicadas a los nodos esclavo.
- El maestro es un punto único de fallo.
- Las lecturas pueden hacer en cualquier nodo (por lo general desde los esclavos para liberar de carga al nodo maestro).
- Está indicado para trabajos de lectura intensiva, ya que la capacidad de lectura se puede escalar en horizontal añadiendo nodos esclavo.
- No está indicado para trabajos de escritura intensiva porque todas ellas se realizan en el (único) nodo maestro.
- No está indicado para casos en los que la consistencia de las lecturas sea una necesidad, ya que se puede realizar una lectura en un nodo esclavo antes de que le llegue un dato ya escrito en el maestro.

Par-a-par (*peer-to-peer*):

- No hay un nodo maestro sino que todos (*peers*) están al mismo nivel jerárquico.
- Al no haber maestro, no hay un punto único de fallo.
- Se puede escribir y leer en todos los nodos.
- Cada vez que se escribe en uno la escritura se replica posteriormente en los demás.
- Puede producir inconsistencias de escritura si se modifica a la vez un mismo dato en distintos nodos.
- Puede a su vez emplear dos estrategias para gestionar la concurrencia:
  - Pesimista:
    - Emplea bloqueos para asegurar que un registro no se puede modificar a la vez en dos nodos distintos.
    - Disminuye la disponibilidad durante el tiempo que tarda en levantarse el bloqueo.
  - Optimista:
    - No hay bloqueos, permitiendo inconsistencias durante el tiempo que las escrituras tardan en propagarse.
    - La base de datos se mantiene en todo momento disponible.

Uno de los modos más empleados para conseguir redundancia de datos dentro de una máquina (típicamente en servidores) es instalar 2 o más discos conectados mediante una tecnología de virtualización del almacenamiento llamada RAID.

Ten en cuenta que en este caso no nos referimos a redundancia gracias a usar un clúster, sino al método común para tener redundancia en máquinas individuales.

Puedes ver más información sobre lo que significa replicación de datos en el siguiente enlace:

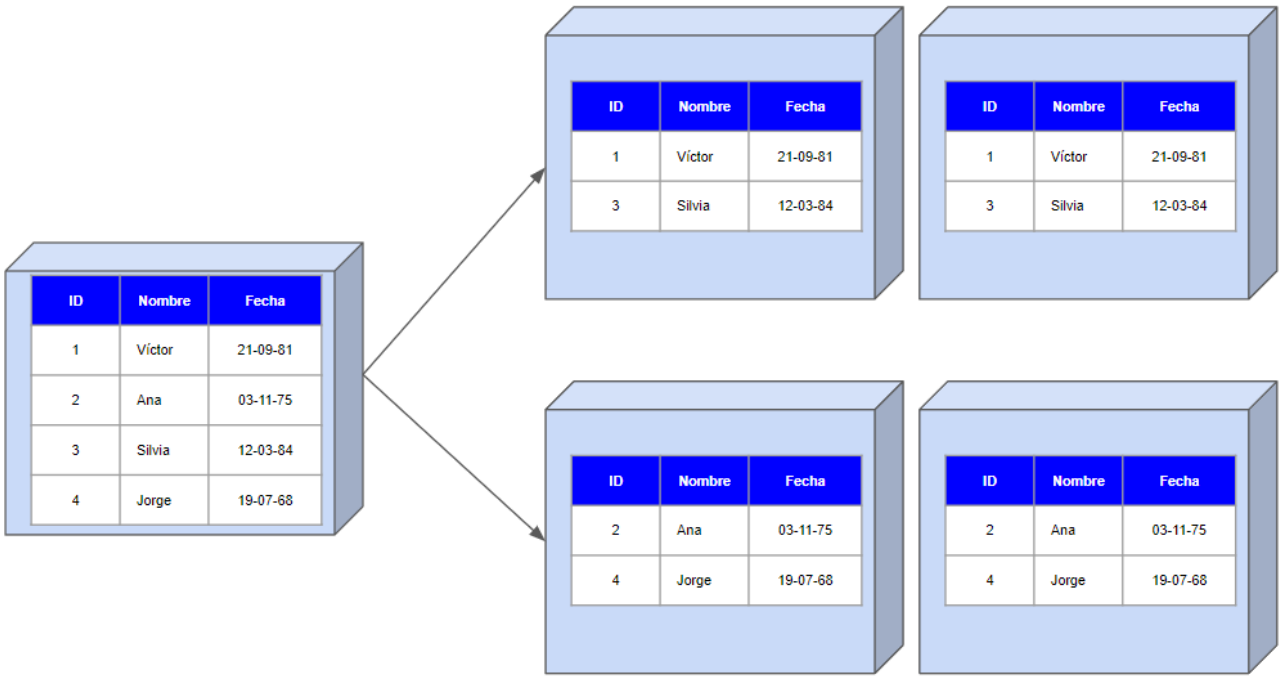
Replicación (informática)

En este enlace puedes ver más información sobre lo que es un punto único de fallo:

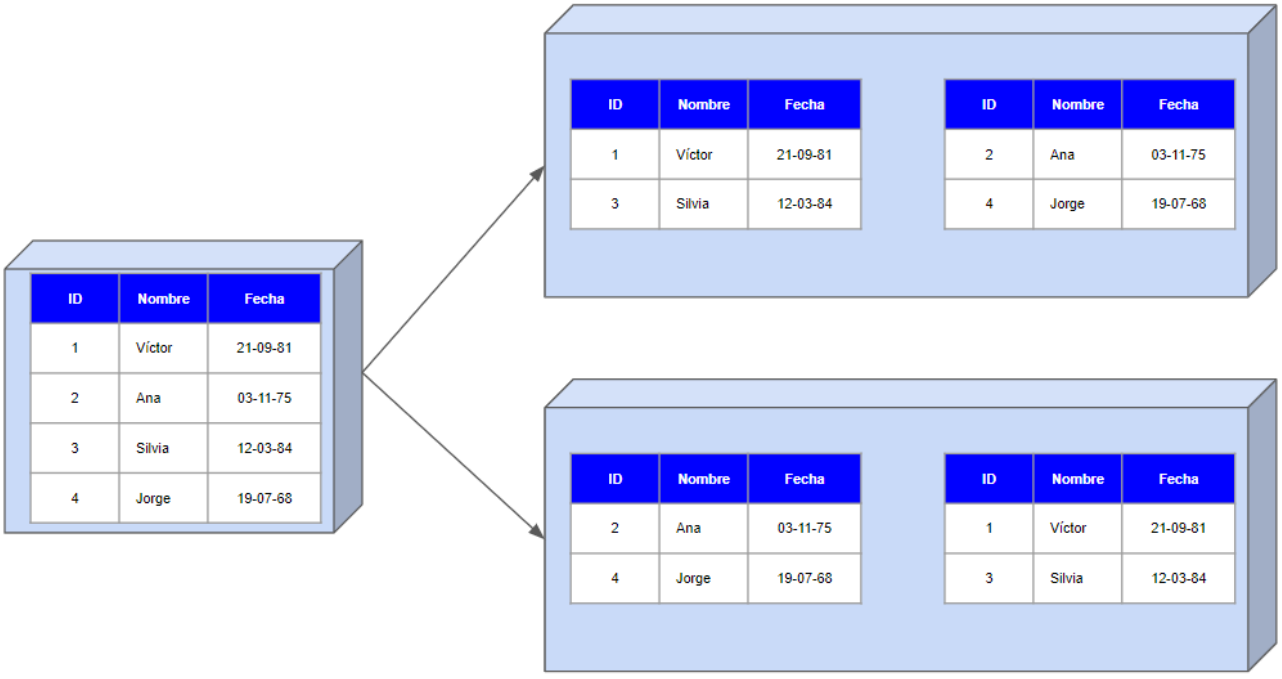
Punto único de fallo

2.1.3.- Sharding con Replicación.

Ejemplo 1:



Ejemplo 2:



Una base de datos distribuida puede implementar tanto ***sharding*** como **replicación**, todo ello funcionando al mismo tiempo.

De ese modo, cada *shard* está a su vez replicado en distintos nodos, con lo que se consigue tolerancia a fallos junto a disponibilidad y escalabilidad.

Hay que tener en cuenta que en cada nodo podemos mantener más de una réplica de shard. Por ejemplo un nodo puede contener la réplica que hace de maestro del shard 1 y una de las réplicas esclavo del shard 2.

Ya que la replicación puede emplear dos distintas estrategias para gestionar la concurrencia, aparecen por lo tanto dos posibilidades al añadir *sharding*.

***Sharding* con replicación maestro-esclavo:**

- Para cada *shard* habrá un maestro y determinado número de esclavos.
- El nodo que contiene un maestro de *shard* es un punto único de fallo para ese *shard*.
- Las escrituras que afectan a un *shard* se realizan en su maestro y después se propagan a los esclavos.

***Sharding* con replicación par-a-par:**

- Para cada *shard* hay varias réplicas al mismo nivel de jerarquía distribuidas por el cluster.
- Al no haber maestros no hay ningún punto único de fallo.
- Las lecturas y escrituras que afectan a un *shard* pueden realizarse en cualquier de sus réplicas (lo cual puede producir los mismos problemas que ya vimos al hablar de la replicación de forma individual).